软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3358−3374 [doi: 10.13328/j.cnki.jos.007221] [CSTR: 32375.14.jos.007221]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

代价敏感的指纹可变哈希布谷鸟过滤器*

李 猛, 罗文啟, 戴海鹏, 王瀚橙, 顾 荣, 陈贵海

(南京大学 计算机学院, 江苏 南京 210023)

通信作者: 戴海鹏, E-mail: haipengdai@nju.edu.cn; 陈贵海, E-mail: gchen@nju.edu.cn

摘 要: 布谷鸟过滤器是一种空间高效的近似成员资格查询数据结构, 在网络系统中被广泛应用于网络路由、网络测量和网络缓存等. 然而, 传统的布谷鸟过滤器设计并未充分考虑在网络系统中, 部分或全部查询集合已知的情况, 以及这部分查询具有代价的情况. 这导致现有的布谷鸟过滤器在该情况下性能无法达到最优. 为此, 设计了指纹可变哈希布谷鸟过滤器 (VHCF). VHCF 提出了指纹可变哈希技术, 感知已知的查询集合及其代价, 通过为每个哈希桶搜索最优指纹哈希函数, 从而大幅降低误判代价. 随后, 每个哈希桶的最优指纹哈希函数会被独立地记录进入每个哈希桶内的哈希索引单元. 此外, 提出了一种单哈希的技术用于降低引入指纹可变哈希技术导致的额外计算开销, 还对 VHCF 的操作复杂度和误判率进行了理论分析. 最后, 实验和理论结果都一致表明, VHCF 在保证查询吞吐量相当的情况下, 取得了比现有布谷鸟过滤器及其变种都要低的误判率. 特别的, 在保持指纹长度相同的情况下, VHCF 只需为每个哈希索引单元分配 1–2 比特, 即可相比标准布谷鸟过滤器降低误判率 12.5%–50%.

关键词: 布谷鸟过滤器; 代价敏感; 集合成员查询; 误判率

中图法分类号: TP393

中文引用格式: 李猛, 罗文啟, 戴海鹏, 王瀚橙, 顾荣, 陈贵海. 代价敏感的指纹可变哈希布谷鸟过滤器. 软件学报, 2025, 36(7): 3358–3374. http://www.jos.org.cn/1000-9825/7221.htm

英文引用格式: Li M, Luo WQ, Dai HP, Wang HC, Gu R, Chen GH. Cost-sensitive Variable Hashing-fingerprint Cuckoo Filter. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3358–3374 (in Chinese). http://www.jos.org.cn/1000-9825/7221.htm

Cost-sensitive Variable Hashing-fingerprint Cuckoo Filter

LI Meng, LUO Wen-Qi, DAI Hai-Peng, WANG Han-Cheng, GU Rong, CHEN Gui-Hai

(School of Computer Science, Nanjing University, Nanjing 210023, China)

Abstract: A cuckoo filter is a space-efficient approximate membership query data structure, widely used in network systems for applications such as network routing, network measurement, and network caching. However, the traditional design of cuckoo filters has not adequately considered the scenario in network systems where some or all queries in the collection are known, and these queries come with associated costs. This limitation results in the suboptimal performance of existing cuckoo filters in such situations. To address this, the variable hashing-fingerprint cuckoo filter (VHCF) has been developed. VHCF introduces variable fingerprint hashing technology, taking into account the known query collection and their associated costs. By searching for the optimal fingerprint hash function for each hash bucket, the overall cost of false positives is significantly reduced. In addition, this study proposes a single-hash technology to reduce the additional computational overhead caused by the variable-hash technology. A theoretical analysis of the operational complexity and false positive rate of VHCF is also provided. Finally, experimental and theoretical results both demonstrate that VHCF achieves a significantly lower false positive rate than existing cuckoo filters and their variants while ensuring comparable query throughput. Specifically, VHCF only needs to allocate 1–2 bits for each hash index unit, which can reduce the false positive rate to 12.5%–50% of the original.

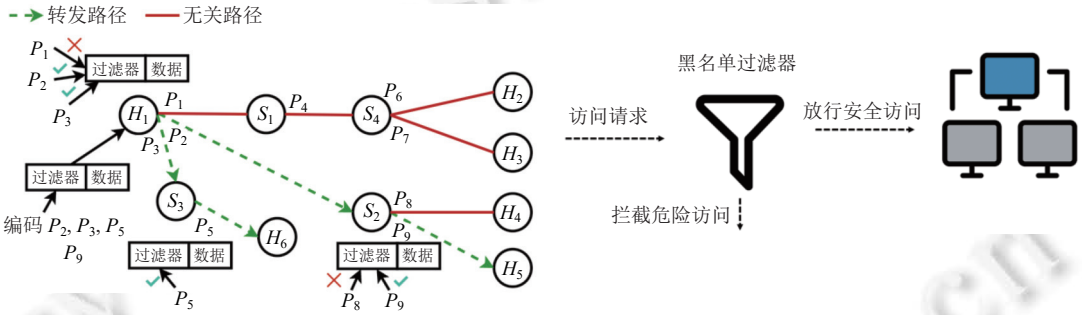
Key words: cuckoo filter; cost-sensitivity; membership testing; false positive rate

过滤器作为一种在空间和时间上高效的数据结构, 被广泛应用于数据挖掘[1−3]、高性能网络[4−13]和数据库系统[14−16]等领域, 用于处理近似集合成员查询问题. 作为经典的布隆过滤器[17]的改进方案, 布谷鸟过滤器 (cuckoo

* 收稿时间: 2023-09-14; 修改时间: 2024-05-08; 采用时间: 2024-05-11; jos 在线出版时间: 2024-07-03
CNKI 网络首发时间: 2024-07-05

filter)[18]由于更加优秀的性能, 近些年被越来越多的采用[4,7,14,16,19−23]. 例如, 在数据挖掘中, 它被用于过滤掉不必要的候选数据, 以提高数据挖掘效率; 在网络过滤中, 它被用于过滤掉不必要的流量, 以提高网络访问请求的性能; 在数据库查询中, 它被用于过滤掉不必要的记录, 以减少昂贵的磁盘查询操作. 布谷鸟过滤器是一种基于哈希表的数据结构, 它保证了插入和查询操作的时间复杂度都为 O(1). 布谷鸟过滤器的基本原理是将元素映射到哈希表的两个位置, 并将这些元素的指纹存储在其中一个哈希桶中. 当需要查询一个元素是否在集合中时, 布谷鸟过滤器会检查哈希表中对应的桶中是否存在该元素的指纹. 如果存在, 那么该元素在集合中; 否则, 该元素就不在集合中. 需要注意的是, 由于哈希表中存储的是元素的指纹而非元素本身, 布谷鸟过滤器会有一定的误判率, 也就是会将未存储在过滤器内的元素误判为在过滤器内. 但是, 传统的布谷鸟在设计时假设每个元素出现的次数或者误判的代价是均匀的. 因此, 当查询元素的频率非均匀甚至非常倾斜时, 过滤器的误判率会增大. 例如, 如果一个元素 A 重复了 1 000 次, 而另一个元素 B 只出现了 1 次, 那么误判元素 A 带来的误判代价就会更大. 为了更好地解释这个问题, 本文给出两个实际系统中的例子.

案例 1. 如图 1(a) 所示, 在数据中心网络多播中, 一些最新的研究工作[9]提出将转发路径或端口存进过滤器. 然后, 在数据包转发过程中, 交换机会检查某个端口是否在过滤器中, 以决定是否将数据包转发至该端口. 然而, 由于过滤器的误判, 这些数据包可能会被错误地转发至一些额外的端口, 从而导致网络带宽的浪费和占用. 尽管存在误判, 但错误地转发至不同端口所造成的网络带宽占用是不同的. 例如, 由于 P_4 端口连接了更多的网络端口 (P_6 和 P_7), 因此错误地将流量转发至 P_4 端口会比错误地转发至 P_8 端口带来更多的网络带宽占用.



(a) 网络多播中利用过滤器实现数据包转发 (b) 网络入侵检测系统快速判定恶意网络流量

图 1 不同元素遭遇过滤器误判带来的代价不相同

案例 2. 如图 1(b) 所示, 网络入侵检测系统[24]在检测网络访问请求时, 为了提高检测速度, 会将黑名单内的 IP 地址编码进过滤器, 从而能快速判断恶意网络流量. 然而, 一个明显的问题是, 一些正常的网络流量被误判后, 也必须经过额外的检查过程, 这必然降低正常访问的速度. 特别是, 一些重要的访问请求或常用应用的流量一旦被误判, 整体的延迟就会大大增加.

考虑到布谷鸟过滤器的优秀性能和丰富的操作, 已有许多基于布谷鸟过滤器的研究, 例如提升布谷鸟过滤器的性能[19,25]; 增加更多的功能, 如支持扩容[20,26]和支持数据库中复杂的操作符[14]. 然而, 这些研究都忽略了查询元素集合在一些网络应用中是可以获取的. 这里的查询集合是指在查询过程中的负元素 (即不在过滤器内的元素) 的信息是可以获取的, 且这些元素在被过滤器误判时的代价并不相同. 据我们所知, 目前还没有布谷鸟过滤器的设计考虑到元素误判代价的不同. 最相关的研究是 Xie 等人[27]提出的一种基于布隆过滤器的变种 HABF, 但由于布谷鸟过滤器和布隆过滤器的结构和操作均不相同, 因此无法直接将 HABF 的方法应用过来.

对于构建代价敏感的布谷鸟过滤器问题, 主要挑战有以下两点: (1) 布谷鸟过滤元素主要通过随机哈希实现, 所以很难直接控制哈希函数值来避免某个元素误判; (2) 过滤器设计中加入的复杂操作会大幅降低过滤器查询速度, 限制布谷鸟过滤器的应用场景.

针对上述问题, 本文提出了一种基于指纹可变哈希布谷鸟过滤器 (variable hashing-fingerprint cuckoo filter,

VHCF). 我们注意到布谷鸟过滤器中元素被误判的根本原因在于一个桶内元素的指纹发生了冲突. 为了解决这个问题, 指纹可变哈希的基本思想是通过改变产生元素的指纹信息的哈希函数, 使得指纹可以改变, 避免一些代价过高的元素被误判. 为了实现指纹可变的目标, 一个简单的解决方法是为每个指纹均匀地分配独立的空间, 用于记录产生指纹的哈希函数. 然而, 均匀地分配空间会降低整体的空间利用率. 这是因为很多元素不会发生冲突, 或者发生冲突的代价很低, 为这样的元素分配独立的空间不会起到作用. 另一方面, 一些元素发生冲突的代价比较大, 给它们分配的空间不够还会造成整体的误判代价较高. 因此, 在指纹可变哈希布谷鸟过滤器中, 本文采取了以哈希桶为单位来选择最优的哈希函数, 即在同一个桶内的元素使用相同的哈希函数产生指纹, 并且产生指纹的哈希函数会被每个哈希桶统一记录.

为了更好地说明, 我们以图 2(b) 为例解释具体的构造步骤. 如图 2(a) 描述了前置步骤, 即根据应用自身需求划分集合 P 和集合 N, 集合 P 中的元素会被插入到过滤器中, 而集合 N 中的元素会在过滤器中进行查询. 然后, 如图 2(b) 所示, 再构造一个辅助布谷鸟哈希表 (cuckoo hash table), 借助辅助布谷鸟哈希表我们可以有针对性地优化每个哈希桶所使用的哈希函数, 以降低总体误判代价. 最后, 我们可以得到如图 2(c) 所示的优化后的哈希函数, 其中每个哈希桶内哈希索引单元内的数字表示该桶内元素使用产生指纹的哈希函数编号.

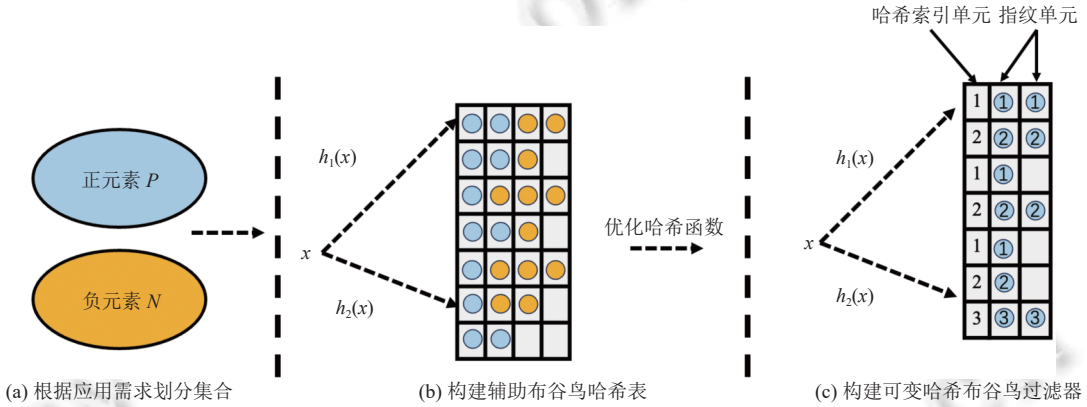


图 2 构建代价敏感的指纹可变哈希布谷鸟过滤器框架图

最后, 在多个数据集上广泛测试了本文提出的指纹可变哈希技术的布谷鸟过滤器, 结果显示, 在使用相同的空间下, 本文的方法相比传统布谷鸟过滤器, 整体的误判率平均降低至原来的 20%–33%.

本文的创新点包括: (1) 在问题定义层面, 引入了代价敏感近似集合成员查询问题, 指出查询集合的全部或部分是可以获取的, 并且每个查询拥有一个误判代价来反应其重要程度. (2) 在方法层面, 本文提出了一种全新的指纹可变哈希技术, 优化每个桶的指纹生成函数, 降低误判代价. (3) 在理论分析和实验层面, 本文提供了严密的理论分析和丰富的实验证明, 从多角度论证了提出方法的优越性.

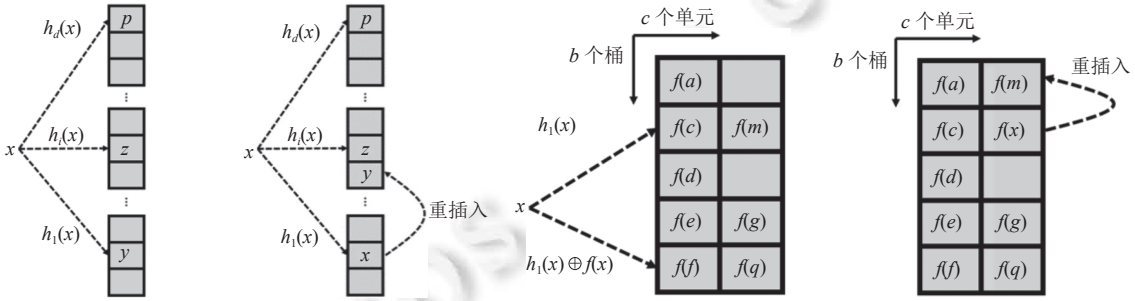
本文第 1 节介绍基础知识以及布谷鸟过滤器相关工作. 第 2 节详细介绍指纹可变哈希布谷鸟过滤器的设计思路和操作方法, 包括如何通过改变产生元素的指纹信息的哈希函数来避免元素冲突, 以及如何选择最优的哈希函数. 第 3 节给出指纹可变哈希布谷鸟过滤器的性能理论分析. 第 4 节通过对比实验验证所提过滤器的有效性. 第 5 节总结全文并对指纹可变哈希布谷鸟过滤器的优势和不足进行讨论, 并提出未来的研究方向.

1 布谷鸟过滤器研究背景和相关工作

考虑到现有基于布谷鸟过滤器的相关工作都沿用布谷鸟哈希表的结构, 并在此基础上提升布谷鸟过滤器的性能及增加其功能, 因此本文将从基础的布谷鸟哈希表和布谷鸟过滤器开始介绍相关工作.

1.1 布谷鸟哈希表

布谷鸟哈希表是一种高效的哈希表实现, 可以支持单个数据的插入和查询操作. 如图 3(a) 所示, 布谷鸟哈希表由 d 个子哈希表组成, 其中的每个子表由 b 个桶组成并且每个桶有 c 个单元. 此外, 这些 d 个哈希子表分别使用 d 个哈希函数 $h_1, h_2, \ldots, h_d$. 每个哈希函数都会将插入的元素均匀地映射到对应哈希表中的一个桶内. 给定元素 x, 它只会被放置在 d 个子表中的某一个, 即在第 i 个子表的桶 $h_i(x)$ 内. 同时, 由于一个哈希桶内最多有 c 个单元用于存储元素, 因此当有超过 c 个元素映射到同一个桶内时, 需要将冲突的元素移动到其他子表内的桶中. 例如, 如图 3(b) 中所示, 元素 x 在每个子表内映射的对应位置都已经被占用了, 因此需要将第 1 个子表内冲突的元素 y 先踢出子表, 然后重新插入下一个子表. 这里如果下一个子表还是被占用, 上述过程可能要重复多次直到有一个空位置被找到. 如果重复次数过多后依旧没有空位置, 那就表明当前哈希表已经装满了, 需要重新扩容以容纳更多的元素.



(a) 布谷鸟哈希表 (b) 布谷鸟哈希表重插入 (c) 布谷鸟过滤器 (d) 布谷鸟过滤器重插入

图 3 布谷鸟哈希表和布谷鸟过滤器

布谷鸟过滤器[18]是基于布谷鸟哈希表的近似成员测试数据结构. 与布谷鸟哈希表不同的是, 布谷鸟过滤器不存储整个元素, 而只存储元素的指纹信息, 从而可以使用更少的空间. 由于存储的是指纹信息, 因此布谷鸟过滤器会产生误判, 将不在过滤器内的元素误认为在过滤器内, 但是这个误判率是很低的, 并且可以被理论证明是有界的.

如图 3(c) 所示, 布谷鸟过滤器由单个哈希表组成, 这个哈希表由 b 个桶组成, 每个桶内包含 c 个单元用以存储元素指纹信息. 当需要插入一个元素 x 进布谷鸟过滤器时, 每个元素都被两个哈希函数映射到 2 个桶中, 每个桶最多可以存储 c 个元素. 为了获取两个桶位置以及元素指纹信息, 布谷鸟过滤器仅仅使用两个哈希函数 $h_1(x)$ 和指纹哈希函数 $f(x)$. 具体而言, 如果元素 x 的指纹为 $f(x)$, 则其两个桶位置 idx_1 和 idx_2 由以下两个哈希函数给出:

$$\begin{cases} idx_1 = h_1(x), \\ idx_2 = h_1(x) \oplus f(x). \end{cases}$$

这里只使用两个哈希函数的优点在于, 只需给定一个桶的位置和一个指纹, 就可以通过当前的桶位置和指纹信息进行位异或操作, 从而确定另一个桶的位置. 如图 3(d) 所示, 布谷鸟过滤器的插入过程和布谷鸟哈希表类似, 也需要经历重插入过程, 从而尽可能提升可容纳的元素数目. 此外, 当指纹的长度为 e 时, 标准布谷鸟过滤器的误判率可以由公式 $\frac{c}{2^{e-1}}$ 给出.

1.2 布谷鸟过滤器的改进工作

为了拓展和提升布谷鸟可用性和功能, 前人已经提出了多种基于布谷鸟过滤器的变种. 这些变种基本关注 3 类问题: 提升布谷鸟过滤器的访问性能, 支持动态扩容以及支持更复杂的查询.

第 1 类变种工作主要着眼于提升布谷鸟过滤器的性能. 其中, Morton 过滤器[19]通过有偏插入机制和稀疏数据压缩等技术, 直接提升了布谷鸟过滤器的插入、查询和删除等操作的性能. Vacuum 过滤器[25]关注过滤器在内存访问时缓存的特性, 设计了由小范围到大范围逐步提升的重插入机制, 提升了布谷鸟过滤器的插入和查询性能. 此

外, 为了支持高速的插入操作, Fu 等人[21]提出了垂直布谷鸟过滤器, 通过给每个元素赋予更多的候选位置减少了布谷鸟哈希表中的重插入操作, 从而提高插入性能. Reviriego 等人[7]提出了完美布谷鸟过滤器使得布谷鸟过滤器在使用一定额外空间的条件下完全避免了误判. 在动态场景下, Mitzenmacher 等人[28]提出了自适应布谷鸟过滤器, 可以动态地修改误判元素的指纹, 避免指纹冲突造成误判. 在处理有明显分布特征的数据集时, Kraska 等人[29]提出可以利用机器学习技术显著降低过滤器所需要的空间. 基于这个基本想法, 为了进一步降低过滤器的误判率, PLBF[30]和 SF[31]陆续被提出. 需要注意的是, 这些学习型过滤器只能在有明显分布的数据上才能发挥作用, 在随机性较大的数据集上误判率会急剧上升, 甚至弱于标准布谷鸟过滤器.

第 2 类变种主要是为了支持动态扩容, 即在动态场景下数据动态增删时调整过滤器空间. 首先, Chen 等人[22]提出了动态布谷鸟过滤器 DCF. DCF 由多个相同结构的子布谷鸟过滤器组成, 并且可以动态增加或减少子布谷鸟过滤器来适应工作负载的动态变化. 此外, Luo 等人[32]利用一致性哈希技术设计了索引独立布谷鸟过滤器 I2CF. I2CF 借用了一致性哈希函数的思想, 全局维护一个一致性哈希环, 并为每个元素分配 k 个候选桶, 从而使得桶的总数可以动态地增加或减少. 为了进一步减轻扩容导致的额外开销, Zhang 等人[20]提出了对数动态布谷鸟过滤器 LDCF. LDCF 设计了一种可扩展的多级树结构来连接多个子布谷鸟过滤器, 并进一步地根据元素指纹的前缀来定位元素在树结构的具体位置, 从而将查询复杂度从线性降低到对数复杂度. Fu 等人[33]拓展了 I2CF 等人的思想, 基于跳跃一致性哈希提出了跳跃滤波器, 可以同时实现随数据基数线性增长的空间开销以及常数级数据访问时开销. 进一步地, Wang 等人[26]为了支持更为细粒度的扩容, 借用线性哈希函数的思想细粒度地扩展布谷鸟过滤器的空间, 并提出了 Bamboo 过滤器.

第 3 类变种主要是基于布谷鸟过滤器拓展支持更多的查询功能. Ting 等人[14]提出了条件布谷鸟过滤器, 用以大幅减少计算数据库中连接操作时需要处理的数据量. Gu 等人[15]基于布谷鸟过滤器构建了一个数据摘要, 用来辅助估计云计算系统中动态负载下需要的缓存大小.

2 代价敏感的指纹可变哈希布谷鸟过滤器设计

在进入具体的技术细节解释之前, 本文首先定义需要研究的问题.

2.1 问题定义

代价敏感近似集合成员查询问题. 给定集合 P 和集合 N, 且 $N \cap P = \varnothing$, 在近似地判定某个元素在集合 P 和集合 N 中时, 如果集合 N 中某个元素 x 如果被误判在 P 中, 其误判的代价为 $C(x)$, 要求最小化总体的误判代价. 为了解决该问题, 本文提出构建一个过滤器 F 近似回答某个元素是否在 P 中, 最小化误判的代价 $\sum_{x \in N} F(x) \cdot C(x)$, 其中 $F(x) = 1$ 表示过滤器 F 判定元素 x 在集合 P 中. 需要强调的是, 这里的“代价敏感”是指在解决近似集合成员查询问题时, 在考虑元素有误判代价的情况下, 调整过滤器结构, 从而最小化总体的误判代价. 表 1 列出了本文用到的符号.

表 1 指纹可变哈希布谷鸟过滤器 VHCF 中参数含义

参数	含义	参数	含义
P	需要存储在过滤器内的元素集合	b	VHCF内包含的哈希桶个数
N	需要避免误判的元素	c	VHCF单个哈希桶可存储的指纹单元数
$C(x)$	元素 x 的误判代价	d	VHCF单个哈希桶内哈希索引单元比特数
$f(x)$	元素 x 的指纹哈希函数	e	VHCF单个哈希桶内指纹单元的指纹长度

上述问题定义中涉及两个关键步骤还需要进一步说明: (1) 如何划分集合 P 和集合 N; (2) 如何获取误判代价 $C(x)$. 首先, 我们关注如何划分集合 P 和集合 N. 需要强调的是, 集合 P 和集合 N 是根据具体应用的实际需求自然划分的. 为了更好地说明, 我们以无状态网络多播应用为例, 集合 P 对应需要进行广播的节点, 而集合 N 对应不需要广播的节点. 集合 P 和集合 N 的划分完全取决于网络广播应用的需求. 同样, 在网络黑名单过滤的例子中, 集合

P 和集合 N 的划分由需要拦截的高风险网站决定. 其次, 我们关注如何获取误判代价 $C(x)$. 这里的误判代价 $C(x)$ 也是根据特定应用场景要求来设定的. 例如在网络多播场景中, 不同级别的端口所连接的机器数量不同, 导致过滤器误判时转发给不同端口带来的额外开销是有差异的. 我们可以根据每个端口不同的误判开销来为其分配一个特定误判代价. 而在网络黑名单中, 我们可以根据恶意网站访问的流量设置其误判代价.

代价敏感的近似集合成员查询问题与传统的近似集合成员查询问题的定义有两点不同. 一是本文假设集合 N 的信息是可以获得的, 即不在 P 集合内的元素信息是可知的. 传统问题定义中并没有假设集合 N 内元素信息是可获取的, 但本文的假设是基于实际应用的. 例如, 在网络多播机制中需要转发的端口集合和不需要转发的端口集合信息都是明确可知的, 对应到本文中集合 N 和集合 P 的信息都是可以获取的. 此外, 集合 N 的信息在许多网络应用中也是可以获取的[27,29]. 二是本文假设集合 N 的元素被误判的代价是不相同的, 并应该被区别对待. 这个假设同样可以在现实应用中找到对应的例子. 以网络多播问题为例, 数据中心网络拓扑中不同级别的端口连接的机器数量是不一样的, 因此错误转发给不同端口带来的冗余流量和带宽占用是显著不同的. 同样, 在检测恶意网站时, 一些知名的恶意网站占用的网络流量远远超过其他普通恶意网站, 正确区分这些恶意网站带来的代价是必需的.

当然, 为了解决上述问题, 一个简单的解决方案是直接存储或压缩存储集合 P, 但显然占用的空间较大, 这在一些对空间要求较高的应用中是难以应用的, 例如在网络交换机上或多播数据包头部. 考虑到苛刻的空间要求, 因此可行的方案是利用空间效率较高的过滤器来近似地解决上述问题. 然而, 传统过滤器在设计时忽略了上述两个问题的假设, 导致误判率较高. 为了解决上述的代价敏感近似集合成员查询问题, 本文设计了一种指纹可变哈希布谷鸟过滤器.

2.2 基本思路

为了解决代价敏感近似集合成员查询问题, 本文设计了指纹可变哈希布谷鸟过滤器, 目标是降低总体的误判代价, 特别是避免高代价元素被误判. 为了实现这个目标, 本文观察到元素被误判的原因是不同元素映射到了同一个桶内, 并且在一个桶内拥有相同的哈希指纹. 因此, 为了避免特定元素被误判, 可行的方法就是改变映射的哈希桶的位置或者改变冲突元素的指纹信息. 但是, 考虑到本文需要保持映射到桶的随机性, 从而保持哈希表能尽量均匀地装载更多的元素, 也就是说, 无法直接改变映射的哈希桶的哈希函数, 因此本文转而考虑修改冲突的元素的指纹信息.

然而, 直接修改指纹会导致被修改的元素无法被正确查询到. 因此, 本文的方法是不直接修改指纹, 而是修改产生指纹的哈希函数. 例如, 当元素 x 与误判代价较高的元素 y 发生冲突时, 即 x 和 y 在同一个桶内, 并且它们的指纹完全相同, 即 $f(x) = f(y)$, 其中 f 是产生指纹的哈希函数. 本文将哈希函数 f 修改为另一个哈希函数 f_1, 这样元素 x 和元素 y 冲突的概率就会大大降低.

那么, 是否需要为每个冲突的元素都去修改指纹哈希函数. 考虑到过滤器是一个空间非常高效的结构, 如果为每个元素都去修改指纹哈希函数, 记录的成本就会很高, 而且重复计算不同的哈希函数也会极大地降低过滤器的查询吞吐率. 因此, 本文并不会单独修改每个元素, 而是只为误判代价高的元素修改指纹函数. 考虑到误判代价高的元素占比并不多, 即每个哈希桶内的误判代价较高的元素数量有限, 一个更好的做法是按照桶为单位调整哈希函数. 因此, 本文为每个哈希桶单独分配一个空间, 用于记录该哈希桶内修改后的哈希函数.

2.3 结构设计

为了实现上述目标, 本文设计了如后文图 4 所示的指纹可变哈希布谷鸟过滤器. 指纹可变哈希布谷鸟过滤器的结构与布谷鸟过滤器相似, 都是由多个哈希桶组成. 但不同的是, 在每个哈希桶内部, 除了用于记录元素指纹信息的指纹单元, 还有一个额外的指纹哈希索引单元, 用于记录该桶内产生指纹的哈希函数的编号. 此外, 不同于标准布谷鸟过滤器的是, 本文中用于确定桶位置和生成指纹的哈希函数是相互独立的.

2.4 指纹可变哈希布谷鸟过滤器的查询过程

指纹可变哈希布谷鸟过滤器的查询过程与标准布谷鸟过滤器的查询过程类似, 都是检查映射到的两个哈希桶内是否包含与查询元素相同的指纹, 唯一不同之处在于每个桶内产生指纹的哈希函数是不同的, 因此需要根据每

个桶内的哈希索引单元存储的哈希函数重新计算元素的指纹, 并在桶内检查是否存在相同的指纹. 需要注意的是, 由于根据哈希索引单元内的哈希函数重新计算指纹会引入额外的计算代价, 可能会降低查询的吞吐率. 为解决该问题, 本文设计了一种单哈希技术, 可以大幅降低哈希函数计算的开销.

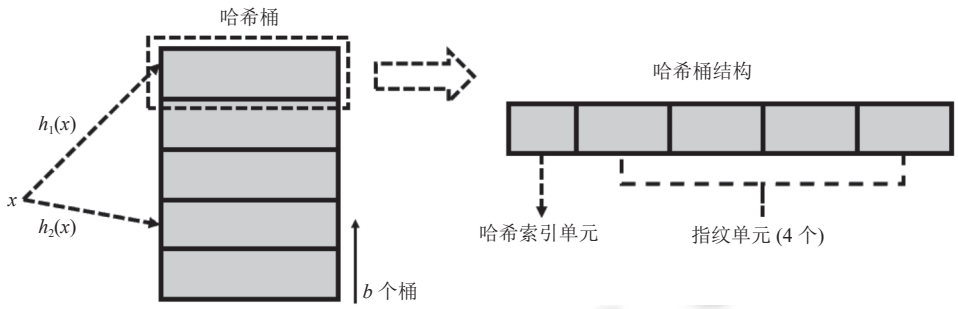


图 4 指纹可变哈希布谷鸟过滤器结构

下面首先介绍指纹可变哈希布谷鸟过滤器 (VHCF) 的查询流程, 然后介绍如何利用单哈希技术降低哈希函数计算的开销.

2.4.1 查询流程

如算法 1 所示, VHCF 的查询操作由两部分组成: (1) 定位到两个候选映射的哈希桶 (步骤 1); (2) 在桶内搜寻是否有相同的指纹 (步骤 2–5). 具体而言, 步骤 1 首先计算两个候选哈希桶的位置 $h_1(x)$ 和 $h_2(x)$. 步骤 2 是根据候选桶内的哈希索引单元计算元素指纹信息, 步骤 3 就是在候选桶内搜寻是否有相同的指纹. 如果有相同的指纹, 步骤 4 就返回查询的元素在 VHCF 内部, 否则步骤 5 就返回元素不在 VHCF 内.

算法 1. 指纹可变哈希布谷鸟过滤器 VHCF 的查询操作.

输入: 待查询元素 x;
输出: 元素 x 是否在 VHCF 内.

1. 计算映射到的两个哈希桶, 也即 $h_1(x)$ 和 $h_2(x)$;
2. 从哈希索引单元内获取哈希函数编号 i 并计算出指纹信息 $f_i(x)$;
3. **If** 桶 $h_1(x)$ 和 $h_2(x)$ 包含由指纹 $f_i(x)$;
4. **Return** x 在 VCHF 内;
5. **Return** x 不在 VHCF 内;

2.4.2 利用单哈希技术降低哈希计算开销

由于每个桶的指纹哈希函数是可变的, 导致在访问不同桶的时候, 需要反复计算不同的指纹哈希函数. 为了更好地说明上述问题, 本文在图 5(a) 中给出一个例子. 在图 5(a) 中所示, 不同的桶用于产生指纹的哈希函数不同, 在第 1 个桶内产生指纹的哈希函数是 $f_1(x)$, 而在第 2 个桶产生指纹的哈希函数则是 $f_3(x)$. 由于每个桶内的指纹哈希函数不同, 因此每次访问一个桶的时候都需要重新计算新的指纹, 这就带来了额外计算开销, 而标准的布谷鸟过滤器只需要计算一次. 为了解决这部分额外的计算开销, 本文引入了一种单哈希技术降低由于引入指纹可变哈希导致的额外计算代价.

单哈希技术的基本原理是哈希函数复用. 为了更好地说明上述基本原理, 本文给出了具体的例子. 如图 5(b) 所示, 给定一个元素利用哈希函数产生指纹时, 实际产生的随机哈希值比较长, 一般可以达到 64 比特或者更长. 然而, 布谷鸟过滤器中的指纹长度却比较短, 例如 6–12 比特. 因此, 本文可以复用这些较长的哈希值划分为多个较短的哈希指纹, 如图 5(b) 下半部分所示. 总而言之, 通过使用单哈希技术复用哈希值, 可以规避或缓解指纹可变哈希技术带来的计算开销.

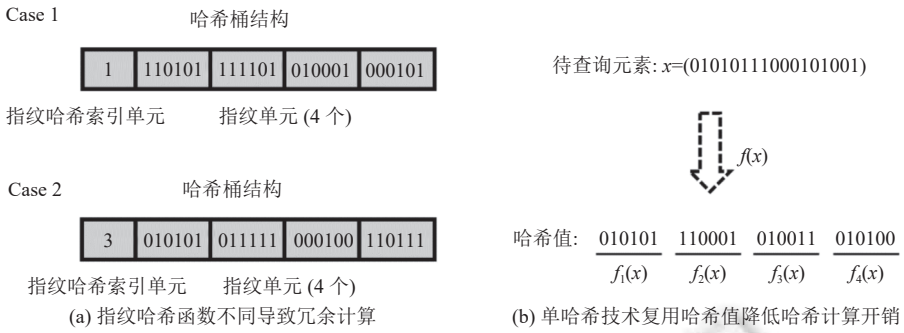


图 5 利用单哈希技术降低哈希计算开销

2.5 指纹可变哈希布谷鸟过滤器的构造过程

本节介绍如何构造 VHCF, 以实现在给定数据集下整体的误判代价最低. 在集合 N 中, 元素被误判的根本原因是其指纹与过滤器中的某个指纹发生冲突. 本文提出了改变产生指纹的哈希函数的方法, 以减少冲突的发生. 此外, 指纹可变哈希布谷鸟过滤器的每个桶内还包含一个额外的哈希索引单元, 存储当前桶内用于产生指纹的哈希函数编号. 在确定了基本方案之后, 下一个重要问题是如何选择最优的哈希函数, 以降低整体的误判代价. 为了解决这个问题, 本文将哈希函数的选择问题形式化为一个优化问题, 并提出了算法 2 来降低整体的误判代价.

算法 2. 指纹可变哈希布谷鸟过滤器 VHCF 的构造算法.

输入: 元素全集 U;

输出: 元素 x 是否在 VHCF 内.

1. 根据应用需求将元素分别划分至集合 P 和集合 N 内;
2. 构造辅助布谷鸟哈希表, 再将集合 P 中每个元素 x 插入布谷鸟哈希表, 即桶 $h_1(x)$ 或 $h_2(x)$;
3. 将集合 N 中的元素同时标记插入布谷鸟哈希表的两个哈希桶, 即桶 $h_1(x)$ 和桶 $h_2(x)$;
4. **For** 哈希桶 B_i:
5. 将映射到 B_i 内的 P 集合内元素记为 P_i, 并将集合 N 内映射到桶 B_i 内的内元素记为 N_i;
6. 初始化当前桶内误判代价 $C_i=+\infty$, 初始化当前桶最优哈希函数编号为 I;
7. **For** 指纹函数集合 $\{f_1, f_2, \ldots, f_{2^d}\}$ 中每一个哈希函数 f_i:
8. 计算选择哈希函数 f_i, 时的误判代价 $\sum_{x\in N_i} C(x)\cdot Sign\left(\sum_{y\in P_i} f_i(x)==f_i(y)\right)$;
9. 如果 $C<C_i$, 则将当前的函数编号记录进入 I 内部;
10. 将 I 的编号记录进入指纹可变哈希布谷鸟过滤器对应桶的哈希索引单元;
11. 依次生成 P_i 内每个元素的指纹并记录进入 VHCF 对应桶内;

2.5.1 VHCF 构造过程

如图 2 所示, 构造 VHCF 的主要步骤分为 3 步: (1) 明确集合 P 和 N 内的元素; (2) 构建辅助布谷鸟哈希表; (3) 求解最优哈希函数配置并生成指纹可变哈希布谷鸟过滤器 VHCF. 为了更清晰地展现构建过程, 完整的构建过程见算法 2. 本文按照 3 个部分逐步解释整个的构建过程.

第 1 部分是划分不同集合内的元素, 见算法 2 的步骤 1, 根据应用的需求将不同的元素划分至对应集合. 例如, 网络多播机制中, 需要把转发的端口划分到集合 P 内, 而将其余端口划分进入集合 N 内. 下一步是构建辅助布谷鸟哈希表, 见算法 2 的步骤 2、3, 需要分别将集合 P 内的元素和 N 内的元素分别映射到辅助的布谷鸟哈希表中. 这里需要注意的是, 集合 P 内的元素只会放置在两个候选桶的中一个, 而集合 N 的元素则会同时插入两个映射的

哈希桶内. 这里 N 内两个桶内都要插入的原因是因为元素在判定是否在过滤器内部的时候需要检查两个哈希桶内是否有相同的指纹, 因此只有同时插入到两个映射的哈希桶内, 才能在后续步骤中规避集合 N 中的元素被误判. 注意, 由于最终的过滤器内不会存储集合 N 内的元素信息, 集合 N 内的元素插入辅助布谷鸟哈希表时采用标记插入的方式, 即挂载在这个桶内的一个指定空间, 这样就避免了集合 P 内的元素竞争辅助布谷鸟哈希表内哈希桶存储空间. 第 3 部分是核心步骤, 需要根据辅助哈希表内每个桶内元素情况求解最优哈希函数. 为了解决这个问题, 本文首先将这个问题形式化为一个优化问题: 寻找一个指纹哈希函数 f_i, 使得当前桶内的误判代价最低, 也即

$$\begin{cases} \min C(B_i) = \sum\limits_{x\in N_i} C(x)\cdot Sign\left(\sum\limits_{y\in P_i} f_i(x) == f_i(y)\right), \quad i\in[0,2^d-1] \\ \text{s.t.} \quad Sign(x)=\begin{cases} 0, & x<0 \\ 1, & x\geqslant 0 \end{cases} \end{cases} \tag{1}$$

为了解决上述问题, 本文只需要遍历所有的候选指纹哈希函数, 然后选择误判代价最低的指纹哈希函数即可.

为了更好地说明上述过程, 本文将具体的步骤形式化地呈现在算法 2 的步骤 4–11. 首先, 步骤 4–11 会循环遍历所有的哈希桶, 然后在循环过程中依次优化每个桶内的哈希函数. 步骤 5 将集合 P 和 N 中映射到当前哈希桶 B_i 内的元素分别记录为集合 P_i 和 N_i. 步骤 6 初始化临时变量误判代价 C_i 为最大整数值以及最优哈希函数编号 I 为 −1. 步骤 7–9 计算每个候选指纹哈希函数的误判代价, 并将误判代价最低的指纹哈希函数记录进入变量 I 内. 如步骤 10、11 所示, 在获取最优的哈希函数编号 I 后, 为当前桶内的 P_i 元素使用最优的指纹哈希函数生成指纹, 然后连同最优哈希函数编号和指纹一一对应地插入指纹可变哈希布谷鸟过滤器的相同的桶内. 构造结束.

2.5.2 优化构造过程

在构造指纹可变哈希布谷鸟过滤器时, 为哈希桶 B_i 选择最优的哈希函数时, 需要遍历映射到该哈希桶内所有的元素, 即 P_i 和 N_i. 这里集合 N_i 内的元素数量可能会远大于集合 P_i 内的元素数量, 这就会极大地拖累整体的构造时间. 为了应对集合 N_i 内元素过多导致构造时间过长的问题, 本文提出一种优化后的构造过程. 每个哈希桶内只保存最多 K 个 N_i 集合内的元素. 为了保证整体的误判代价最低, 本文只保存 K 个误判代价最高的元素. 为了实现上述目标, 可以先将集合 N 内的元素按照误判代价从高到低排序依次映射进入辅助布谷鸟哈希表内, 但每个桶内只保留误判代价最高的 K 个元素, 进入下一轮用于搜寻最优哈希函数, 这直接降低了构造开销.

2.5.3 支持频繁更新的场景

尽管 VHCF 的设计主要面向静态集合场景, 但通过推迟和周期性重构, 在动态场景下 VHCF 的性能可以依旧不逊于标准布谷鸟过滤器. 具体的方案如下: 当新元素插入 VHCF 的桶内时, 我们可以选择不立即进行重构, 而是直接使用哈希索引单元内的哈希函数计算指纹, 然后将其直接插入. 通过这样的策略, 新插入元素的误报率与标准布谷鸟过滤器完全一致. 最后, 等到元素变动比例超过一定阈值时, 才进行整体重构, 从而大幅降低误报率. 这一策略有效地平衡了性能和动态变化的需求.

3 理论分析和优化

为了更好理解本文提出的指纹可变哈希布谷鸟过滤器的性能, 本节从查询复杂度、构造复杂度、误判率 3 个方面展开进行理论分析. 下面先从查询复杂度入手, 然后再分析误判率.

3.1 复杂度分析

3.1.1 查询时间复杂度分析

指纹可变哈希布谷鸟过滤器查询过程和标准布谷鸟过滤器的查询过程相似, 都需要使用两个哈希函数访问两个候选哈希桶, 然后检查两个哈希桶内是否含有匹配的指纹信息. 不同的是, VHCF 需要根据哈希索引单元重新计算指纹信息. 因此, 指纹可变哈希布谷鸟过滤器 VHCF 的查询复杂度为:

$$\mathrm{O}(2\cdot(1+c))=\mathrm{O}(c) \tag{2}$$

考虑到此处的参数 c 是个常数, 因此上述查询复杂度可以简化为 O(1).

3.1.2 构造时间复杂度分析

指纹可变哈希布谷鸟过滤器的构造过程主要涉及将集合 P 和 N 内的元素先插入辅助布谷鸟哈希表, 然后根据每个哈希桶内元素频率分布情况选取最优的哈希函数. 首先, 布谷鸟哈希表的插入复杂度平摊下来是 O(1), 这就使得插入集合 P 内的元素的复杂度为 O(|P|), 而插入集合 N 内元素的复杂度为 O(|N|). 此外, 在搜寻最优哈希函数时, 需要遍历所有候选指纹哈希函数, 然后找到最优的哈希函数. 由于候选的哈希函数数目取决于每个桶内哈希索引单元的比特数 d, 也即 2^d 个候选指纹哈希函数. 因此, 搜索最优哈希函数复杂度为:

$$\mathrm{O}\left(\sum_{i=0}^{b-1} N_i \cdot 2^d \cdot c\right) = \mathrm{O}(2 \cdot |N| \cdot 2^d \cdot c) = \mathrm{O}(|N| \cdot 2^d) \tag{3}$$

将所有步骤的复杂度合并后可知, 最终的构造复杂度为 $\mathrm{O}(|P| + |N| \cdot (2^d + 1))$.

3.1.3 空间复杂度分析

对于空间复杂度, 过滤器的空间复杂度通常用 BPK (bits-per-key) 指标衡量, 即每个元素占用的比特数目, 该参数通常是由用户指定. 因此, 在给定 BPK 和总元素个数的情况下, 空间复杂度为 $p \cdot BPK$. 此外, 空间复杂度也可以表示为 $b \cdot (d + c \cdot e)$, 其中 b 是 VHCF 内包含的哈希桶个数, c 是 VHCF 单个哈希桶可存储的指纹单元数, d 是 VHCF 单个哈希桶内哈希索引单元比特数, e 是 VHCF 哈希桶内指纹单元的指纹长度.

复杂度分析总结: 在时间复杂度分析方面, VHCF 的查询复杂度是常数级别, 与标准布谷鸟过滤器查询复杂度相同. 但由于引入哈希索引单元, 略慢于标准布谷鸟过滤器. 在构造时间复杂度方面, VHCF 的构造复杂度正比于集合 P 和集合 N 的总大小, 同样由于引入了最优哈希函数搜索的步骤, VHCF 的构造时间复杂度高于标准布谷鸟过滤器.

3.2 误判率分析

本文先计算在不考虑哈希索引单元的情况下的误判率, 即映射到相同哈希桶内的同时并且映射的桶内有相同的指纹信息. 因此, 由于本文只考虑静态场景下已经映射到同一个桶内的元素, 只需要考虑在同一个桶内和 c 个长度为 e 的指纹冲突的概率, 误判率表示为:

$$p = \frac{c}{2^e} \tag{4}$$

进一步地, 本文继续考虑引入了哈希索引单元后的误判率, 为了分析方便, 本文将集合 N 的元素内映射进入哈希桶 B_i 内的元素集合记为 N_i, 并简记 $|N_i| = n$. 由于引入了指纹哈希单元, 我们允许一个单元最多使用 2^d 个哈希函数, 并且只选择误判率最低的哈希函数. 因此, 上述问题可以转化成为求解 2^d 个哈希函数中误判元素个数最少的哈希函数对应的误判个数, 即求解下列式子的期望:

$\min(h^1, h^2, \ldots, h^{2^d})$, h^i 指代第 i 个哈希函数误判的元素个数.

这里, 首先可以发现 h^i 都是独立同分布的变量, 为了求解方便, 引入一个新变量 Y 且

$$Y = \min(h^1, h^2, \ldots, h^{2^d}) \tag{5}$$

首先求解 Y 的累计概率分布, $\Pr(Y \leqslant i) = 1 - \Pr(h^1 > i, h^2 > i, \ldots)$. 由于 h^i 是独立同分布变量, 因此该式子可以简化为:

$$\Pr(Y \leqslant i) = 1 - \Pr(h^1 > i) \cdot \Pr(h^2 > i) \cdot \ldots \cdot \Pr(h^{2^d} > i) = 1 - \Pr(h^1 > i)^{2^d} \tag{6}$$

下面, 继续求解 $\Pr(h^1 > i)$, 即单个哈希函数误判个数的累计概率分布函数. 通过分情况讨论, h^1 有如下几种情况: (1) $h^1 = 0$ 的概率为 $p_0 = (1-p)^n$; (2)$h^1 = 1$ 的概率为 $p_1 = (1-p)^n p_1 = \binom{n}{1} p(1-p)^{n-1}$; (3) 拓展至一般情况时, 误判个数为 i 的概率, $p_i = \binom{n}{i} p^i (1-p)^{n-i}$. 因此, 可以得知:

$$\Pr(h^1 > i) = 1 - \sum_{j \in [0,i]} p_j \tag{7}$$

因此, 将 $\Pr(h^1 > i)$ 代入下列式子, 可得:

$$\Pr(Y \leqslant i) = 1 - \left(1 - \sum_{j\in[0,i]} p_j\right)^{2^d} \tag{8}$$

此处, 由于 Y 是离散变量且为整数, 根据 $\Pr(Y \leqslant i)$ 的解析式, 可以求解 $\Pr(Y = i)$ 的解析式:

$$\begin{aligned}\Pr(Y = i) &= \Pr(Y \leqslant i) - \Pr(Y \leqslant i-1)\\ &= 1 - \left(1 - \sum_{j\in[0,i]} p_j\right)^{2^d} - 1 + \left(1 - \sum_{j\in[0,i-1]} p_j\right)^{2^d}\\ &= \left(1 - \sum_{j\in[0,i-1]} p_j\right)^{2^d} - \left(1 - \sum_{j\in[0,i]} p_j\right)^{2^d}\end{aligned} \tag{9}$$

根据期望的定义, 可得:

$$\begin{aligned}E(Y) &= \sum_{i\in[0,n]} i\cdot\Pr(Y = i)\\ &= \sum_{i\in[0,n]} i\cdot(\Pr(Y \leqslant i) - \Pr(Y \leqslant i-1))\\ &= n\cdot\Pr(Y \leqslant n) - \sum_{i\in[0,n-1]} \Pr(Y \leqslant i)\\ &= n - \left(1 - (1 - \sum_{i\in[0,0]} p_i)^{2^d}\right) + \ldots + \left(1 - \left(1 - \sum_{i\in[0,n-1]} p_i\right)^{2^d}\right)\\ &= \left(1 - \sum_{i\in[0,0]} p_i\right)^{2^d} + \ldots + \left(1 - \sum_{i\in[0,n-1]} p_i\right)^{2^d}\end{aligned} \tag{10}$$

这里先检查最简单的情况 $d = 0$, 哈希索引单元的空间为 0 比特.

$$\begin{aligned}E(Y) &= \left(1 - \sum_{i\in[0,0]} p_i\right)^{2^0} + \ldots + \left(1 - \sum_{i\in[0,n-1]} p_i\right)^{2^0}\\ &= n - (np_0 + (n-1)p_1 + \ldots + p_{n-1})\\ &= n - \left(n\binom{n}{i} p^i(1-p)^{n-i}\right)\\ &= n - n\left(\binom{n}{i} p^i(1-p)^{n-i}\right)\\ &= n - n(1-p)\left(\binom{n-1}{i} p^i(1-p)^{(n-1)-i}\right)\\ &= n - n(1-p)\\ &= np\end{aligned} \tag{11}$$

下面检查 $d = 1$ 的情况, 即哈希索引单元的空间为 1 比特时:

$$E(Y) = \left(1 - \sum_{i\in[0,0]} p_i\right)^{2^1} + \ldots + \left(1 - \sum_{i\in[0,n-1]} p_i\right)^{2^1} \tag{12}$$

这里需要简单分析 $\sum_{j\in[0,i]} p_j$ 函数的特性. 首先, $\sum_{j\in[0,i]} p_j$ 取值是 $[0,1]$, 并且整体是递增的, 即 $1 - \sum_{j\in[0,i]} p_j$ 整体是递减的. 此外, $\left(1-\sum_{j\in[0,0]} p_j\right)^2 = (1-p_0)(1-p_0)$. 可得, $\left(1-\sum_{i\in[0,0]} p_i\right)^2 = (1-p_0)(1-p_0) \approx (1-p_0)np = (1-p_0)\frac{nc}{2^e}$.

$$E(Y) = \left(1 - \sum_{i\in[0,0]} p_i\right)^{2^1} + \ldots + (1 - \sum_{i\in[0,n-1]} p_i)^{2^1} \leqslant \left(\left(1 - \sum_{i\in[0,0]} p_i\right) + \ldots + \left(1 - \sum_{i\in[0,n-1]} p_i\right)\right)(1-p_0) \leqslant np(1-p_0) \tag{13}$$

同理, 当哈希索引单元的空间增加到 d 个比特时, 可以得到 Y 的期望值如下:

$$\begin{aligned}E(Y)&=\left(1-\sum_{i\in[0,0]}p_i\right)^{2^d}+\ldots+\left(1-\sum_{i\in[0,n-1]}p_i\right)^{2^d}\\&\leqslant\left(\left(1-\sum_{i\in[0,0]}p_i\right)+\ldots+\left(1-\sum_{i\in[0,n-1]}p_i\right)\right)(1-p_0)^{2^d-1}\\&\leqslant np(1-p_0)^{2^d-1}\end{aligned}\tag{14}$$

下面继续讨论 $1-p_0$ 的取值范围. 根据 $1-p_0=1-(1-p)^n=\left(1-\left(1-\frac{c}{2^e}\right)^n\right)\approx\left(1-1+\frac{nc}{2^e}\right)=\frac{nc}{2^e}<1$, 因此, 使用 d 比特的哈希索引空间时, 冲突元素的数量比例至少下降至 $\frac{E(Y)}{np}=\frac{np(1-p_0)^{2^d-1}}{np}=\left(\frac{nc}{2^e}\right)^{2^d-1}$.

误判率分析总结: 针对误判率, 相比标准布谷鸟过滤器, VHCF 的误判率可以显著降低, 并且误判率的下降比例可以被理论证明, 至少为 $\left(\frac{nc}{2^e}\right)^{2^d-1}$.

4 实验分析

本节将展开综合的实验来全面评估指纹可变哈希布谷鸟过滤器的性能. 首先介绍实验配置, 包括实验平台、对比算法、评估指标和数据集. 实验主要包括两部分: 一是研究参数变化对于指纹可变哈希布谷鸟过滤器性能的影响; 二是在多个数据集上评估 VHCF 和前人提出的方法并进行对比, 例如误判率、查询和构造时延.

4.1 实验配置

● 实验平台. 使用一台配备了两颗 20 核 40 线程的 Intel Xeon Gold 5218R CPU 以及 64 GB 内存的服务器评估不同算法的性能. 使用 C++实现了指纹可变哈希布谷鸟过滤器 VHCF. 所有实验均使用 GCC 编译器编译并运行在 CentOS 操作系统上. 实验部分汇报的结果都是重复 10 次后取平均值.

● 对比算法. 本文使用两个经典的过滤器和两个最新的数据分布自适应的过滤器进行测试. 4 个对比算法依次是 (1) 标准 Bloom 过滤器 BF[34], 作为一个基准来衡量其他过滤器的性能; (2) 标准布谷鸟过滤器 CF[18], 作为基于布谷鸟过滤器的变种的基准来衡量其他过滤器的性能; (3) 哈希自适应 Bloom 过滤器 HABF[27], HABF 作为 Bloom 的一个变种, 是一个代价敏感的过滤器, 同时也是本文一个主要的对比算法. (4) 自适应布谷鸟过滤器 ACF[28], ACF 是面向在线场景设计的自适应过滤器, 由布谷鸟过滤器和布谷鸟哈希表两部分组成. 当一个元素发生误判时, ACF 会修改引起该元素误判的指纹信息从而降低误判率. 最后, 本文提出的 VHCF 以及所有其他过滤器的测试代码已发布在 https://github.com/Ln7-best/VHCF.

● 性能评估指标. 本文主要在 3 个指标上评估了指纹可变哈希布谷鸟过滤器的性能: (1) 代价加权误判率 (weighted false positive rate, Weighted FPR), 其定义为 $\frac{\sum_{x\in N}F(x)\cdot C(x)}{\sum_{x\in N}C(x)}$, 其中 $F(x)=1$ 表示过滤器 F 判定元素 x 在集合 P 中 (当 $C(x)=1$ 时, 此时代价加权误判率等同于普通 FPR); (2) 查询吞吐量, 查询吞吐量指的是单位时间内过滤器执行查询操作的次数, 单位为百万次每秒 (Mops/s); (3) 构造时延, 本文将其定义为在构造过滤器时, 平均每个元素需要消耗的时间, 单位为纳秒 (ns).

● 测试数据集. 本文在 3 个场景的数据集上评估了算法的性能. 第 1 个场景是在数据中心 Fat Tree 拓扑架构[9]下利用过滤器进行转发. 本文分别生成了 Fat Tree 结构下不同的 k 值对应的误判率. 在 Fat Tree 网络架构中, k 参数是重要的参数, 它表示了网络的规模和复杂性. 具体来说, k 参数定义了 Fat Tree 的端口数量, 也就是每个交换机可以连接的其他交换机或主机的数量. 因此, k 参数直接影响了网络的带宽、容量和延迟等性能指标. 例如, 如果 $k=6$, 那么每个交换机可以连接 6 个其他交换机或主机, 网络的总带宽和容量就会较 $k=4$ 时相应增大. 为了测试方便, 本文随机选择 1 个主机作为发送方, 40% 的主机作为接收方, 并将发送方和接收方之间的最短路径合并成多播的转发路径. 第 2 个场景是恶意流量识别, 本文采用了 CAIDA 数据集[35]进行测试. 这个数据集是从骨干网络流量中收集的真实数据, 包含 700 万个不同的元素. 第 3 个场景是骨干网络路由流量识别, 本文使用了 MAWI[36]数据

集进行测试. 这个数据集包含了从 WIDE 骨干网采集的网络流量信息, 本文以源 IP 目标 IP 作为元素内容, 并从中选取了包含 1 千万条网络流量的子集进行测试. 更为具体的信息, 本文列在了表 2 中. 此外, 在测试 CAIDA 和 MAWI 数据集时, 本文主要关注数据误判代价分布倾斜情况下的不同过滤器的代价加权误判率. 这里为了更好地测试不同的倾斜分布, 本文使用 Zipf 分布, 其通过参数 *skewness* 来衡量倾斜性, 值越大分布越倾斜, 即更少的元素占据了更大的比例.

表 2 实验数据集信息

数据集	不同元素个数	数据类型
CAIDA	约7百万	源目标IP地址
MAWI	约1千万	源目标IP地址

● 参数设置. 为了确保不同算法的公平比较, 本文在实验评估过程中要求所有被比较的算法尽可能使用相同的内存空间. 为了衡量不同过滤器使用的空间大小, 本文使用指标 BPK (bits-per-key), 即每个元素使用的比特数目. 在满足这一要求的基础上, 本文根据每个测试算法的默认参数设置来设定各个算法的参数.

4.2 评估参数影响分析

为了充分探讨在不同情况下, 本文算法性能受哈希索引单元大小的影响, 我们先进行参数实验评估. 由于不同数据集下的实验情况类似, 本文基于 CAIDA 数据集进行测试. 在图 6 中, 本文测试了 3 种不同设置的 VHCF, 即 VHCF-7、VHCF-8、VHCF-9. 其中 VHCF-后的数字标号代表了 VHCF 中哈希桶内指纹的长度 e. 如图 6 所示, 当哈希索引单元的大小 d 增大 (从 1 比特增加到 4 比特), 不同指纹长度配置下的 VHCF 的代价加权误判率 Weighted FPR 均发生了下降, 最多的 VHCF-9 的误判率下降超过 3 个数量级, 最小的 VHCF-7 也下降至 1/16. 由此可见, 哈希索引单元有效地降低了 VHCF 的误判率, 并且整体下降速度随着指纹长度 e 的增加而增加, 哈希索引单元的增大带来的收益明显. 另外一个有意思的发现是, VHCF-7 在使用 4 比特的哈希索引单元时, 误判率比使用了 1 比特和 2 比特哈希索引单元的 VHCF-8 都要低, 这就说明了引入哈希索引单元实际效果比单纯增加指纹长度要好. 同样的情况对于 VHCF-8 和 VHCF-9 也成立.

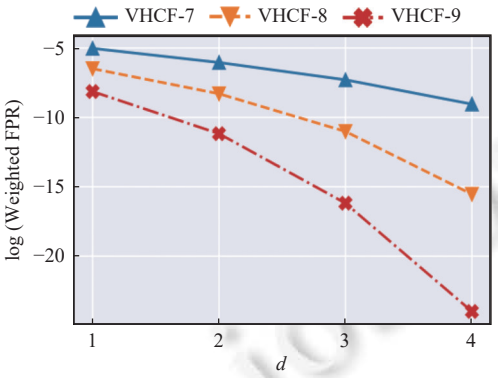
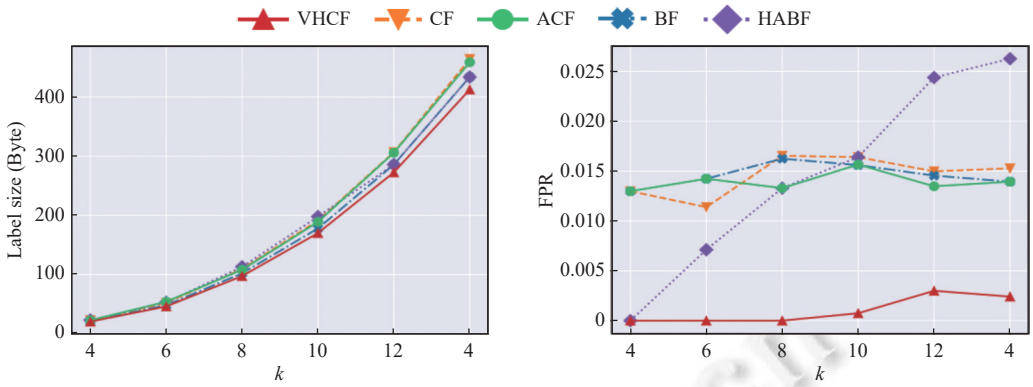


图 6 不同哈希索引单元大小 d 下误判率的变化情况

4.3 评估网络多播场景中误判率

本文评估了在网络多播场景中不同过滤器消耗的空间和对应的误判率的情况. 如图 7(a) 所示, 随着 Fat Tree 的 k 值在不断增长, 所有过滤器占用的空间都在增长. 但是, VHCF 始终保持了最小的空间开销, 并且其空间开销的增长速度也是最缓慢的. 而在图 7(b) 中, 在保持相近的空间大小使用情况下, 可以发现 VHCF 的误判率在不同的 k 值下始终都是最低的, 并且比第二优的 HABF 平均提升 5 倍以上. 对于其他的过滤器, 如 ACF、 CF 和 BF 的误判率则基本相近. 实验结果表明, VHCF 在流量转发场景中有巨大的应用潜力, 可大幅降低转发流量.

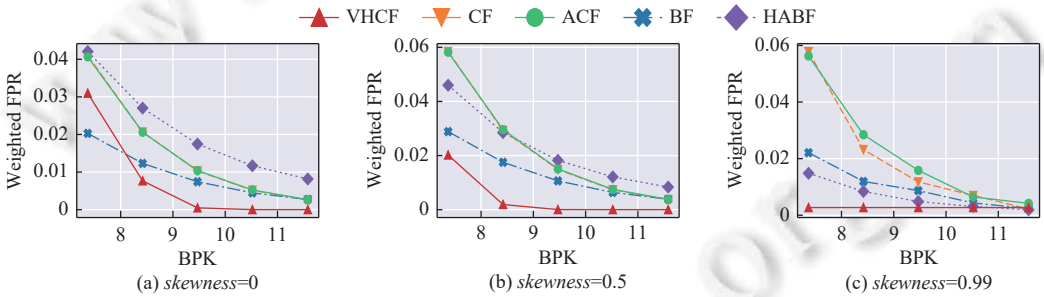


(a) 不同 k 值下包含过滤器的网络包头大小　　(b) 不同 k 值下不同过滤器的误判率

图 7　Fat Tree 网络中不同 k 值下空间占用和误判率对比

4.4　评估网络流量分类场景中误判率

本文评估了网络流量场景中, 在不同的倾斜程度下, 不同过滤器的代价加权误判率随着每个元素分配的比特数量 BPK 的变化. 首先关注 CAIDA 数据集, 如图 8(a) 所示, 在 $skewness = 0$ 也就是均匀分布情况下, VHCF 的误判率在 BPK>8 的情况下是最低的. BF 在空间较小的情况下, 拥有着较低的误判率. 当数据误判代价变倾斜时 ($skewness = 0.5$) , 如图 8(b) 所示, 我们可以发现在各种 BPK 配置下 VHCF 依然保持最好性能, 并且代价加权误判率相较于均匀分布的情况显著降低. 此时, 另外一个代价敏感的过滤器 HABF 的代价加权误判率也开始下降, 但仍明显差于 VHCF. 最后, 当数据分布非常倾斜时 ($skewness = 0.99$) , 如图 8(c) 所示, VHCF 仍有最低的误判率, 且误判率已显著接近 0, 而 HABF 此时的性能已经仅次于 VHCF, 这是因为 HABF 的设计也考虑到了元素误判代价, 但仅在误判代价分布较为倾斜时, 其性能才会较好.



(a) $skewness$=0　(b) $skewness$=0.5　(c) $skewness$=0.99

图 8　CAIDA 数据集不同倾斜程度下, 误判率随每个元素分配的比特数 BPK 变化

MAWI 数据集的实验情况和 CAIDA 数据集类似. 如后文图 9 所示, 除了在 $skewness = 0$ 的情况下 VHCF 的误判率始终是最低的, 并且和其他过滤器性能有显著差异, 比其他最好的过滤器误判率平均降低超过 50%. 此外, HABF 的误判率也随着分布倾斜度增加快速降低. ACF 和 CF 的性能相当并且一直维持着较高的误判率, 这是因为他们的设计中基本不考虑元素分布的信息.

4.5　评估查询吞吐量和构造延迟

本文评估了不同过滤器的查询吞吐量和构造延迟. 如图 10(a) 所示, VHCF 的查询吞吐量超过了 100 Mops/s, 基本接近标准布谷鸟过滤器的查询吞吐量, 远高于 ACF 以及 HABF 的查询吞吐量. 与 ACF 相比, VHCF 采用了单哈希技术, 通过对一个长哈希比特串进行切分得到若干子哈希值, 使得 VHCF 每次查询只需要进行一次哈希计算. 而 ACF 的设计基于布谷鸟哈希表, 同时每个指纹的生成函数都是独立的, 所以无论是确定元素在每个子表中的对应位置还是得到指纹, 都需要引入大量的哈希函数计算, 进而拖慢了查询的速度. 因此, ACF 的吞吐量会远低于

VHCF. 相较于 VHCF, HABF 的设计基于布隆过滤器. 在查询过程中, 需要访问多个随机的比特位, 因此 HABF 需要进行多次哈希计算. 与之不同, VHCF 在查询阶段涉及的是最多 2 个哈希桶, 这导致更少的缓存缺失. 此外, 从获取哈希函数的效率考虑, 在 VHCF 中获取桶中的指纹生成函数仅需要额外读取一次与桶中指纹一起存储的哈希索引单元即可完成. 相较之下, HABF 需要查询额外的辅助结构, 以获取每个元素所需的哈希函数, 这一步骤引入了大量的时间开销. 综合而言, 由于 HABF 在查询过程中可能会引入更多的缓存缺失, 同时需要进行多次复杂的哈希计算, 以及额外的哈希函数获取步骤, 其吞吐量相较于 VHCF 明显较低.

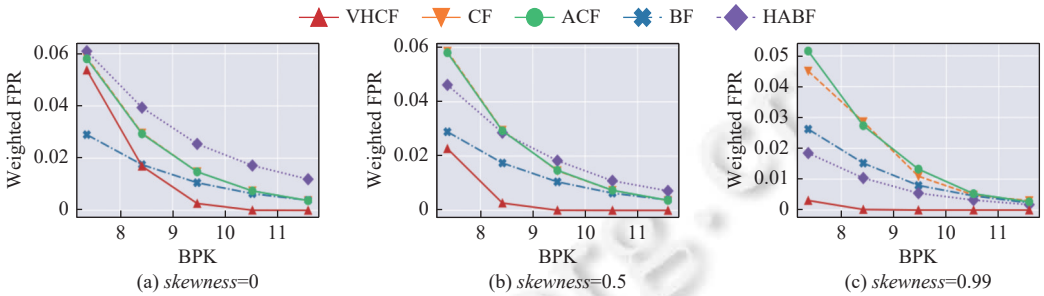


图 9 MAWI 数据集上, 不同倾斜程度下, 误判率随每个元素分配的比特数 BPK 变化

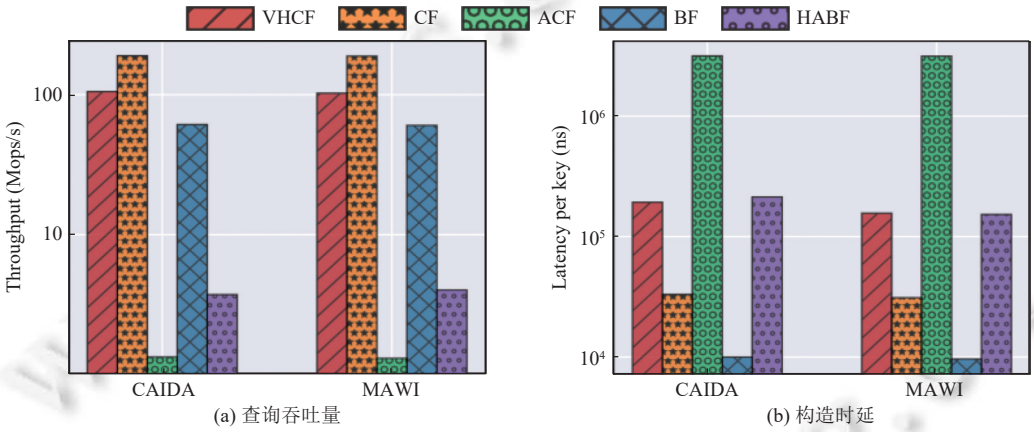


图 10 查询吞吐量和构造时延

与此同时, 所有算法的构造时延如图 10(b) 所示. 由于求取最优哈希函数引入的额外操作, VHCF 的构造速度要慢于两个经典过滤器, 但与自适应过滤器 HABF 基本一致, 远快于 ACF. 实验的所有过滤器中构造速度最快的是 BF, 这是因为 BF 的插入操作仅仅涉及比特位操作. 综合来看, VHCF 的构造时延平均约为 150 μs, 仍可接受.

5 总 结

本研究提出了一种指纹可变哈希技术, 旨在解决静态场景下的代价敏感集合成员问题. 本文设计了一种指纹可变哈希布谷鸟过滤器, 并进一步引入了单哈希技术, 以降低指纹可变哈希所带来的额外哈希计算开销, 显著提升了无状态网络多播等应用场景中的系统性能. 此外, 本文的基本想法是通过利用历史查询信息优化近似索引过滤器的结构设计, 类似的想法还可以复用在其他系统组件如磁盘索引构建, 缓存系统资源控制等, 有望提升更多潜在应用的性能. 最后, 在未来的研究中, 还需要考虑支持动态场景下的代价敏感集合成员问题, 并探索引入更多新技术, 如机器学习等, 从而更为高效地解决此类问题.

References:

[1] Zeng ZX, Xiao RL, Lin XH, Luo TJ, Lin JY. Double locality sensitive hashing bloom filter for high-dimensional streaming anomaly

detection. Information Processing & Management, 2023, 60(3): 103306. [doi: 10.1016/J.IPM.2023.103306]

[2] Saeed ASM, George LE. Fingerprint-based data deduplication using a mathematical bounded linear hash function. Symmetry, 2021, 13(11): 1978. [doi: 10.3390/sym13111978]

[3] Dutta S, Narang A, Bera SK. Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams. Proc. of the VLDB Endowment, 2013, 6(8): 589–600. [doi: 10.14778/2536354.2536359]

[4] Khalid A, Esposito F. Optimized cuckoo filters for efficient distributed SDN and NFV applications. In: Proc. of the 2020 IEEE Conf. on Network Function Virtualization and Software Defined Networks. Leganes: IEEE, 2020. 77–83. [doi: 10.1109/NFV-SDN50289.2020.9289870]

[5] Broder A, Mitzenmacher M. Network applications of bloom filters: A survey. Internet Mathematics, 2004, 1(4): 485–509. [doi: 10.1080/15427951.2004.10129096]

[6] Nikolaevskiy I, Lukyanenko A, Polishchuk T, Polishchuk V, Gurtov A. IsBF: Scalable in-packet bloom filter based multicast. Computer Communications, 2015, 70: 79–85. [doi: 10.1016/j.comcom.2015.05.002]

[7] Reviriego P, Pontarelli S. Perfect cuckoo filters. In: Proc. of the 17th Int'l Conf. on Emerging Networking Experiments and Technologies. ACM, 2021. 205–211. [doi: 10.1145/3485983.3494852]

[8] Fan B, Andersen DG, Kaminsky M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In: Proc. of the 10th USENIX Conf. on Networked Systems Design and Implementation. Lombard: USENIX Association, 2013. 371–384.

[9] Chen ZH, Huang JW, Wang QL, Liu JL, Li ZY, Zhou SW, He ZD. MEB: An efficient and accurate multicast using bloom filter with customized hash function. In: Proc. of the 7th Asia-Pacific Workshop on Networking. Hong Kong: ACM, 2023. 157–163. [doi: 10.1145/3600061.3600062]

[10] Abdennebi A, Kaya K. A bloom filter survey: Variants for different domain applications. arXiv:2106.12189, 2021.

[11] Dai HP, Zhong YK, Liu AX, Wang W, Li M. Noisy bloom filters for multi-set membership testing. In: Proc. of the 2016 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Science. Antibes Juan-les-Pins: ACM, 2016. 139–151. [doi: 10.1145/2896377.2901451]

[12] Laufer RP, Velloso PB, Duarte OCMB. Generalized bloom filters. Technical Report, Rio de Janeiro: COPPE/UFRJ, 2005.

[13] Xie K, Wen JG, Zhang DF, Xie GG. Bloom filter query algorithm. Ruan Jian Xue Bao/Journal of Software, 2009, 20(1): 96–108 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3458.htm [doi: 10.3724/SP.J.1001.2009.03458]

[14] Ting D, Cole R. Conditional cuckoo filters. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 1838–1850. [doi: 10.1145/3448016.3452811]

[15] Gu R, Li SM, Dai HP, Wang HC, Luo YL, Fan B, Basat RB, Wang K, Song ZY, Chen SW, Wang BN, Huang YH, Chen GH. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In: Proc. of the 2023 USENIX Annual Technical Conf. Boston: USENIX, 2023. 467–484.

[16] Li SS, Luo LL, Guo DK, Zhao YW. Stable cuckoo filter for data streams. In: Proc. of the 27th Int'l Conf. on Parallel and Distributed Systems. Beijing: IEEE, 2021. 138–145. [doi: 10.1109/ICPADS53394.2021.00023]

[17] Bloom BH. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422–426. [doi: 10.1145/362686.362692]

[18] Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo filter: Practically better than bloom. In: Proc. of the 10th ACM Int'l on Conf. on Emerging Networking Experiments and Technologies. Sydney: ACM, 2014. 75–88. [doi: 10.1145/2674005.2674994]

[19] Breslow AD, Jayasena NS. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. Proc. of the VLDB Endowment, 2018, 11(9): 1041–1055. [doi: 10.14778/3213880.3213884]

[20] Zhang F, Chen HH, Jin H, Reviriego P. The logarithmic dynamic cuckoo filter. In: Proc. of the 37th Int'l Conf. on Data Engineering. Chania: IEEE, 2021. 948–959. [doi: 10.1109/ICDE51399.2021.00087]

[21] Fu PT, Luo LL, Li SS, Guo DK, Cheng GY, Zhou Y. The vertical cuckoo filters: A family of insertion-friendly sketches for online applications. In: Proc. of the 41st Int'l Conf. on Distributed Computing Systems. Washington: IEEE, 2021. 57–67. [doi: 10.1109/ICDCS51616.2021.00015]

[22] Chen HH, Liao LY, Jin H, Wu J. The dynamic cuckoo filter. In: Proc. of the 25th Int'l Conf. on Network Protocols. Toronto: IEEE, 2017. 1–10. [doi: 10.1109/ICNP.2017.8117563]

[23] Dayan N, Twitto M. Chucky: A succinct cuckoo filter for LSM-tree. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 365–378. [doi: 10.1145/3448016.3457273]

[24] Gebretsadik FG, Nayak S, Patgiri R. EBF: An enhanced bloom filter for intrusion detection in IoT. Journal of Big Data, 2023, 10(1): 102. [doi: 10.1186/s40537-023-00790-9]

[25] Wang MM, Zhou MX, Shi SQ, Qian C. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. Proc. of the VLDB Endowment, 2019, 13(2): 197–210. [doi: 10.14778/3364324.3364333]

[26] Wang HC, Dai HP, Li M, Yu J, Gu R, Zheng JQ, Chen GH. Bamboo filters: Make resizing smooth. In: Proc. of the 38th Int'l Conf. on Data Engineering. Kuala Lumpur: IEEE, 2022. 979–991. [doi: 10.1109/ICDE53745.2022.00078]

[27] Xie RB, Li M, Miao ZY, Gu R, Huang H, Dai HP, Chen GH. Hash adaptive bloom filter. In: Proc. of the 37th Int'l Conf. on Data Engineering. Chania: IEEE, 2021. 636–647. [doi: 10.1109/ICDE51399.2021.00061]

[28] Mitzenmacher M, Pontarelli S, Reviriego P. Adaptive cuckoo filters. ACM Journal of Experimental Algorithmics, 2020, 25(1.1): 1–20. [doi: 10.1145/3339504]

[29] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 489–504. [doi: 10.1145/3183713.3196909]

[30] Vaidya K, Knorr E, Kraska T, Mitzenmacher M. Partitioned learned bloom filter. arXiv:2006.03176, 2020.

[31] Deeds K, Hentschel B, Idreos S. Stacked filters: Learning to filter by structure. Proc. of the VLDB Endowment, 2020, 14(4): 600–612. [doi: 10.14778/3436905.3436919]

[32] Luo LL, Guo DK, Rottenstreich O, Ma RTB, Luo XS, Ren BB. The consistent cuckoo filter. In: Proc. of the IEEE Conf. on Computer Communications. Paris: IEEE, 2019. 712–720. [doi: 10.1109/INFOCOM.2019.8737454]

[33] Fu PT, Luo LL, Guo DK, Zhao X, Li SS, Wang HM. Jump filter: Dynamic sketch design for big data governance. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1193–1212 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6782.htm [doi: 10.13328/j.cnki.jos.006782]

[34] Putze F, Sanders P, Singler J. Cache-, hash-, and space-efficient bloom filters. ACM Journal of Experimental Algorithmics, 2010, 14: 4. [doi: 10.1145/1498698.1594230]

[35] CAIDA. https://www.caida.org/

[36] MAWI working group traffic archive. https://mawi.wide.ad.jp/mawi/

附中文参考文献:

[13] 谢鲲, 文吉刚, 张大方, 谢高岗. 布鲁姆过滤器查询算法. 软件学报, 2009, 20(1): 96–108. http://www.jos.org.cn/1000-9825/3458.htm [doi: 10.3724/SP.J.1001.2009.03458]

[33] 符鹏涛, 罗来龙, 郭得科, 赵翔, 李尚森, 王怀民. 跳跃滤波: 一种面向大数据治理的动态数据摘要设计. 软件学报, 2023, 34(3): 1193–1212. http://www.jos.org.cn/1000-9825/6782.htm [doi: 10.13328/j.cnki.jos.006782]

李猛(1993－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为概率性数据结构, 物联网系统.

王瀚橙(1998－), 男, 博士生, CCF 学生会员, 主要研究领域为数据索引和查询优化.

罗文啟(2001－), 男, 硕士生, 主要研究领域为高性能网络.

顾荣(1988－), 男, 博士, 助理教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据并行处理系统.

戴海鹏(1985－), 男, 博士, 副教授, 博士生导师, CCF 杰出会员, 主要研究领域为物联网, 数据挖掘, 边缘计算, 移动计算.

陈贵海(1963－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为无线网络, 并行计算, 算法设计.